Version 1.3
1/27/1994

Memdebug

Document no TOOL-PM-1

Author: Schmit René
CRP-HT
Internet: Rene.Schmit@crpht.lu

Table of Contents

Memdebug

Product Overview

Memdebug is a utility that helps you debug all memory allocations and frees in your C programs. Use Memdebug to trace memory blocks that have not been freed correctly, to simulate ‘out of memory’ errors and to optimise your `malloc()/free()` sequences. Furthermore, chances are higher that your program will not hang your computer when it performs some illegal memory operation such as freeing an illegal pointer. Of course, all these errors are detected at run-time and logged with as much debugging data as possible.

Memdebug is written in portable ANSI C, and only minor changes to your sources are necessary to use its basic features. Later on, all Memdebug calls may be switched off at compilation time, so that your final product version won’t be affected at all.

Features

Usage Scope

Memdebug supports all ANSI C <`stdlib`> memory management functions: `malloc()`, `calloc()`, `realloc()` and `free()`.

At compilation time, or more precisely during the pre-processor pass, all calls to those functions are changed to calls of Memdebug’s own routines, which call the standard library functions during their execution. When the MEMDEBUG pre-processor symbol is not defined, no substitutions of the standard function calls are made, and all Memdebug-specific function calls are changed to empty statements. Thus, your final program won’t be affected by the presence of Memdebug calls.

Only data concerning memory management is treated by Memdebug. Bugs such as illegal memory references are only partially detected by Memdebug. Memdebug looks out for ‘off by one’ errors at the start and end of a memory block. Furthermore, the content of data may be set to some dummy value by Memdebug’s version of `free()`, which will probably make your program fail when it accesses this memory block later on.

Many errors that are detected won’t have the usual fatal effect. But beware! Don’t use Memdebug as a ‘Bug Preventer’, as eventually, the bugs will turn up again. Moreover, the performances of your program degrade dramatically when Memdebug is used!

Errors

By default, a message will be written to `stderr` for each error that Memdebug detects. `stderr` may be redirected to another text file specified in the `set_MemdebugOptions`() function. Generally, the message itself is preceded by a file name and a line number. It represents the source location where the error occurred.

The following errors are detected by Memdebug:

Memory exhaustion: No more system or Memdebug memory (see later) is available at this point of execution. This error may occur during calls to `malloc()`, `calloc()` and `realloc()`.

Unallocated pointer: A pointer is used that was not obtained by a call to `malloc()`, `calloc()` or `realloc()`. This error may occur during calls to the `realloc()` and `free()` routines.

Spurious `free()` calls: The `free()` routine has been called with the NULL pointer as its argument. According to the ANSI standard, this is no error, and `free()` won't do anything if this happens, but you might consider it bad programming style to `free()` NULL pointers.

Unbalanced allocation/free call number: When the program exits, Memdebug tests whether all allocated blocks have been freed previously. If this is not he case, the number of blocks still allocated are written to `stderr`. More information about the unfreed blocks may be found in the statistical data.

Corrupted memory blocks: As mentioned above, Memdebug tests whether the bytes immediately preceding or following a memory block remain unchanged. If the user writes at these memory locations, an error will be written to `stderr`.

Statistics

During run time, Memdebug gathers an abundance of statistical data concerning memory usage. This information is written to `stdout`, which may be redirected as needed. Following are the main categories of data gathered:

General statistics: They include the number of times a given routine has been called, the current, maximal and overall number of currently allocated blocks and bytes of memory, and some details about the number of errors that occurred.

Next comes an alphabetically sorted list of all the pointer variables that have been used. Indicated are the name of the pointers used when the memory block has been freed, the number of times the block has been freed as well as the overall and average size allocated for this variable.

Follows a list containing all variables that have not been freed. Each line begins with the file name and line number where the memory block was allocated, indicates the size in bytes and the size expression during allocation and the current content of the memory pointed to. Of course, the format of the data is unknown, so it is casted to a string and serves only as an indication. Non-printing characters are replaced by dots, so serial terminals won't get disturbed.

Finally, a list containing a trace of all memory management function calls is printed. This list contains the file name and line number of the function call as well as the name of the pointer variable (if available), the block's size in bytes and the size expression.

Customising Memdebug and Debugging

The user may customise Memdebug's output and it's behaviour during allocation. To do so, he has to insert routine calls into his source programs. As already mentioned earlier, these calls will be wiped out by macro definitions when the MEMDEBUG symbol is not defined during compilation, and the presence of these routine calls will not affect the final program behaviour.

The main customisation routine is the `set_MemdebugOptions`() routine. It serves to select the statistics to be printed and to specify the output files, that's to say the redirection of `stderr` and `stdout`. Furthermore, the user may set a limit in bytes or calls for the allocation routines.

The `print_MemdebugStatistics`() routine may be inserted to explicitly print out the statistics file at a given moment. Thus, the user may make a kind of snapshot of the memory status anywhere in the program. Memdebug won't print out automatically a final statistic report when this routine has been called once.

The `check_MemdebugError()` function returns the number of illegal memory operations that occurred during program execution. This function is especially useful for behind-the-scenes test programs that do no direct output. This is also why no implicit final statistics will be printed.

One last routine, `generate_MemdebugError`(), is used to simulate out of memory errors. The next allocation after this call will fail. With the `set_MemdebugOptions`() routine, you may specify the number of `generate_MemdebugError`() calls that may be done before an error occurs.

User Guide

Using Memdebug in a Program

To produce the MEMDEBUG object code files, you have to compile `memdebug.c`, `memfree.c` and `memalpha.c`. BEWARE: don't define the `MEMDEBUG` symbol, else the utility will end up in an infinite loop!

Hint for UNIX users: with the ar command, create an archive file!

Three operations are necessary to add Memdebug's facilities to a program. First, the <`Memdebug.h`> library header needs to be included into <u>every</u> source file of the program. If the program owns a common header file that is included everywhere, it's simplest to include <`Memdebug.h`> there.

Then, the MEMDEBUG pre-processor symbol has to be defined to enable Memdebug's functions. If possible, you should do this on the compiler call line (using -d or -define or similar). Of course, you may also define (or #undefine) the symbol before every inclusion of <`Memdebug.h`>.

Finally, you will need to include the Memdebug objects file to your link list.

This is all that is needed to get the basic, default information of Memdebug. All errors will be detected by Memdebug, and final statistics will be generated. All output will be send to `stderr` and `stdout`. The internal memory limit is set to infinite, and no call sequence trace is generated (to save some memory). Memory contents are destroyed when a `free()` of a pointer is made.

The first call to a memory management function will start Memdebug. Thus, if no memory function is used, no final statistics will be printed.

Customising and Debugging Functions

```
enum t_Option
{
     c_No = 0,
     c_Yes
};

typedef enum t_Option t_Option;
void set_MemdebugOptions(
                            t_Option  p_GeneralStatistics,
                            t_Option  p_AlphabeticalList,
                            t_Option  p_NotFreeList,
                            t_Option  p_CallSequenceList,

                            t_Option  p_SpuriousFreeList,

                            t_Option  p_PrintContents,
                            t_Option  p_DestroyContents,

                            long      p_GenerateErrorCount,
                            size_t    p_MaximalMemoryAvailable,

                            char*     p_StatisticsFileName,
                            char*     p_ErrorFileName
                       );
```

The set_MemdebugOptions() function changes the options setting of Memdebug. The first four parameters are used to tell Memdebug which statistic listings to print. p_Spuriousfree may be set to c_No if you don't care about free(NULL) calls.

p_PrintContents determines if the contents of an allocated pointer will be printed out in string format. This may be turned off if you don't care about the contents.

When p_DestroyContents is set to c_Yes (the default), the contents of the memory pointed to by a pointer will be replaced by the '@' character before the memory is released. This will probably result in a run-time error when the memory is (illegally) referenced later on.

In p_GenerateErrorCount, you may indicate the number of generate_MemdebugError() calls that may be done before an error is generated. This may be useful if you suspect your program to fail only after the n'th memory allocation. The default error count is 0.

With p_MaximalMemoryAvailable, you may set the maximal amount of memory that Memdebug will allocate at a given time. This parameter will be ignored during subsequent calls of set_MemdebugOptions, and you can't set the memory limitation to 0 bytes (in fact, 0 is used to indicate 'no change in setting'). Note however that system memory may run out before it reaches the limit you indicate, and that Memdebug doesn't care about memory fragmentation. To test real life memory limitations, set this limit to a lower value than the actual memory limit of the target machine!

p_StatisticsFileName and p_ErrorFileName indicate the file names that replace stdout and stderr. If you pass the empty string, stdout and stderr will be used. The previous file(s) will be closed automatically (except, of course, stdout and stderr).

```
void print_MemdebugStatistics     ( void );
```

Calling this function will cause Memdebug to generate a statistic listing, using the currently set options.

```
int  check_MemdebugError          ( void );
```

Returns the number of illegally freed memory, spurious free()s and corrupted blocks. No more implicit statistics will be printed.

```
void generate_MemdebugError       ( void );
```

The next call to malloc(), calloc(), or realloc() will fail when generate_MemdebugError has been called p_GenerateErrorCount times, regardless of the option settings and the size of the block to be allocated.

Limitations and Known Bugs

Memdebug may hide some bugs in your program. Errors such as deallocating an illegal pointer will no more cause your program to hang, but only until Memdebug is turned off again. Thus, you should always inspect the error listing generated by Memdebug.

Memdebug is quite a resource-intensive utility. It will slow down your program considerably (maybe up to a hundred times!!!), and it needs a lot of space to memorise all information. Thus, if your program itself uses much memory, or if you allocate many small blocks, your system may run out of memory, and you can't use Memdebug. But before giving up, try turning off the allocation trace function. This may save a considerable amount of memory!

Due to the slowing down of your application, you should be cautious while debugging real-time applications. Memdebug may mess up all of your timing.

Final statistics printout and data cleanup are done by exit functions. So, if your program also installs exit functions, and your system imposes some limit concerning the number of such functions, you should know that there is one (1) such function in Memdebug. Ensure all additional data cleanup functions are installed AFTER the initialisation of Memdebug, or you might get erroneous 'pointer not freed' messages.

Each redirection of an output uses one file descriptor, which might cause problems with programs using a lot of files.

There might be a very slight chance that there's still some memory bug in Memdebug (such as an unreleased pointer). Unfortunately, debugging Memdebug with Memdebug ends up in an infinite loop…

Besides: beware when compiling the library: don't compile it with the `MEMDEBUG` symbol defined, else your program will end up in an infinite loop. The Memdebug functions themselves call memory allocation functions that, when Memdebug'ed, call again a Memdebug function and so on.

One last note: Never (and in this context, never means *NEVER*) ship a program version that has Memdebug enabled. Anyhow, your customers won't like an application that's that slow and eats up so much memory. Memdebug is not a 'Bug Hiding' tool, but a debugging tool!

Internals

Extensive use is made of the C pre-processor. All function calls are either replaced by macro calls (when Memdebug is enabled), or faded out (in the other case). Furthermore, the pre-defined symbols __FILE__ and __LINE__ are used to get location information. All parameters are passed once with the # prefix, so that Memdebug gets information about the identifiers and size expressions used.

Initialisation of Memdebug is done during the first call of one of Memdebug's routines, including `malloc()`, `calloc()`, `realloc()` and `free()`. An initialisation routine is called each time, but when it has been called once, it will exit immediately on every subsequent call.

Memdebug's last statistic printout and all data cleanup are done at exit time. To achieve this, some exit handlers have been installed, using the atexit() `stdlib` function.

The data structures used by Memdebug are primarily some simple counters to collect usage number and size statistics. The operations performed on these counters are straightforward.

The 'of by one' error checking is done with prefix and postfix sentinels. Instead of allocating a block with the exact size required, Memdebug allocates some additional space (see fig. 1)

Memdebug now fills the two sentinels with a random value, which is also stored in the internal block descriptor (see below). When the block is either realloced or freed, the three values are compared against each other, and an error is generated should one of the sentinels no longer contain its original value.

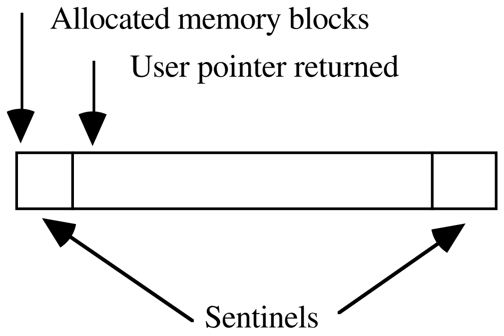

fig. 1

Memorisation and semantic checking of the storage allocator functions required a much more sophisticated data structure. Each memory block is represented by two records. One record contains general information about a variable, while another one contains information about a particular instantiation of the variable. The first one will be called ‘Block Descriptor’, and the second one ‘Call Sequence’.

When a block is allocated, a BlockDescriptor is created, and the pointer to the block is stored in it. A reference to it is stored in an AVL tree called the NotFreedTree, the pointer being used as a key. An AVL tree has been chosen for the following reason: `malloc()` tends to allocate memory in subsequent locations, and using a simple tree would be equivalent to using a list (the tree would degenerate). In consequence, each retrieval of a BlockDescriptor will be accelerated considerable by using an AVL tree, even if insertion and deletion of a node in such a tree are quite costly.

A CallSequence record is allocated next, and in it are stored all data relevant to the allocation of the block (the location, the size and so on). A pointer to the BlockDescriptor is stored into the CallSequence descriptor, and vice versa. This double reference will be used later on during the `free()` operation. Finally, the CallSequence is inserted into the call sequence list (see fig. 2)

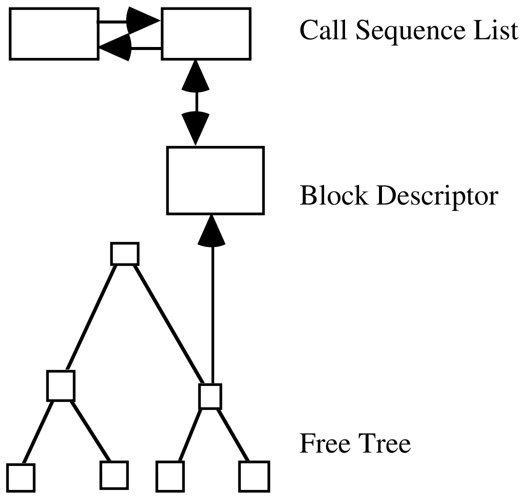

Fig.2

During a `free()` operation, the BlockDescriptor is first looked for in the NotFreedTree. If it isn't found, an illegal deallocation has been attempted, and no further actions are performed. If it is found, the variable that hold the pointer will be looked for in another AVL Tree, the AllocatedTree. In this tree, references to the block descriptors of all already freed variables are stored, the variable identifier serving as key. When the variable is found in this tree, the corresponding BlockDescriptor is updated, and the intermediate block descriptor is freed. If the variable was not used previously in a `free()` call, a reference to the intermediate BlockDescriptor is stored into the tree (see fig. 3).

Furthermore, if the function trace service is enabled, a CallSequence descriptor is appended to the CallSequenceList. In the other case, the `malloc()` descriptor is removed from this list, which saves much memory.

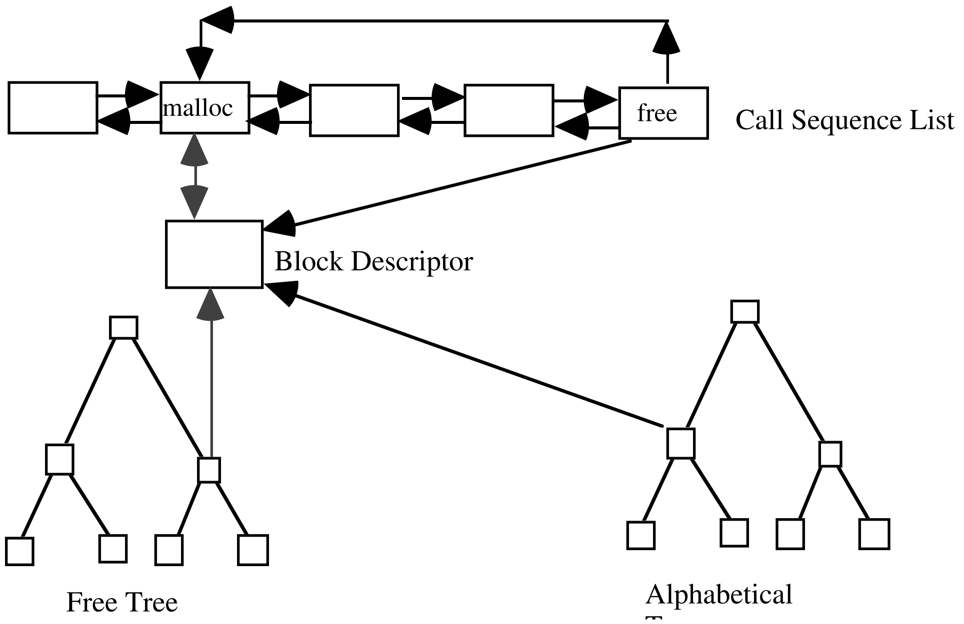

Fig 3.

A `calloc()` call is treated similarly to a `malloc()` call.

A `realloc()` call requires updating the information in the intermediary BlockDescriptor and adding a CallSequence block to the corresponding list. As the pointer to the allocated block is the key in the NotFreedTree and `realloc()` may change this pointer, the reference to the BlockDescriptor is always removed from the tree and inserted again after the reallocation. Furthermore, an additional link is used in the CallSequenceList, connecting a `realloc()` descriptor to its previous `realloc()` or its original `malloc()` or `calloc()` descriptor.

Function Prototype List

```
void generate_MemdebugError      ( void );

void print_MemdebugStatistics    ( void );

int  check_MemdebugError         ( void );

void set_MemdebugOptions(   t_biState     p_GeneralStatistics,
                            t_biState     p_AlphabeticalList,
                            t_biState     p_NotFreeList,
                            t_biState     p_CallSequenceList,

                            t_biState     p_SpuriousFreeList,

                            t_biState     p_PrintContents,
                            t_biState     p_DestroyContents,

                            long          p_GenerateErrorCount,
                            t_MemorySize  p_MaximalMemoryAvailable,

                            char       *  p_StatisticsFileName,
                            char       *  p_ErrorFileName
                        );
```

Bibliography

K&R: `stdlib` for description of `malloc()`,`free()`, `calloc()`,`realloc()`